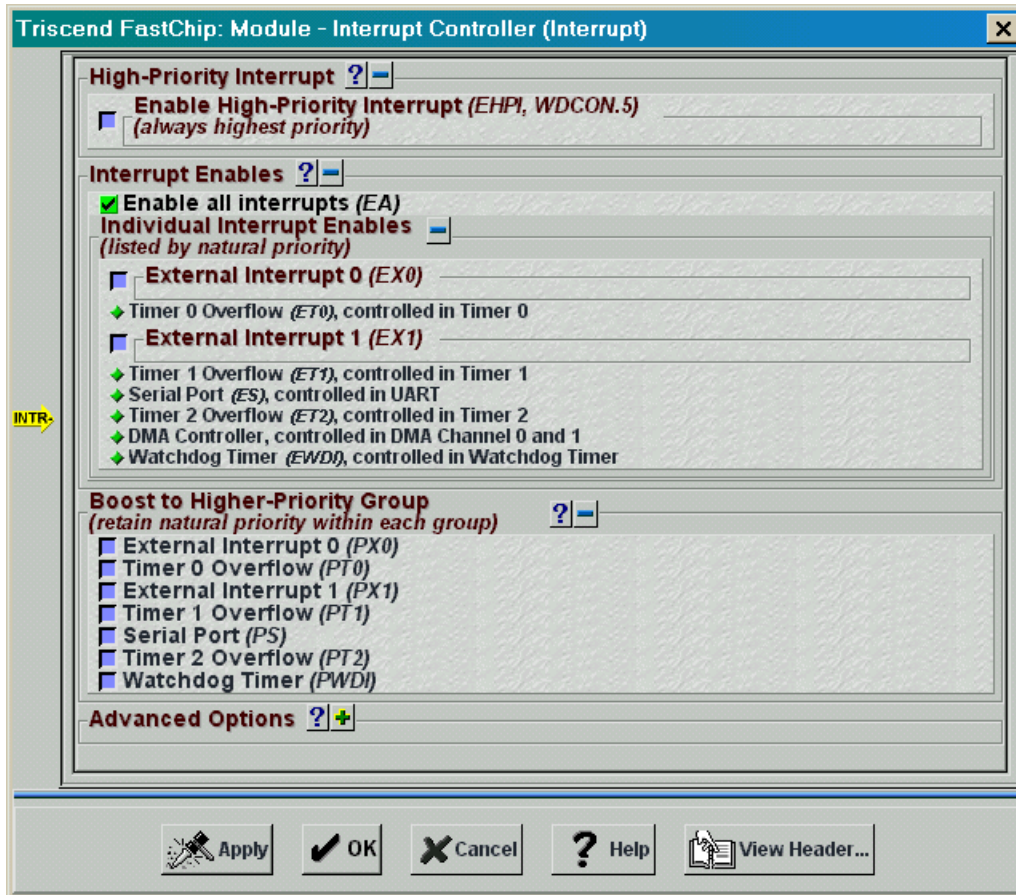


Once the E is displayed, will the O reappear if you return the DIP switch to its OFF position and re-establish the loopback path? Usually not. When you break the loopback path, it will most likely truncate the string of eleven bits comprised of the start bit, eight data bits forming the counter value, the programmable ninth data bit, and the stop bit. If the UART receiver doesn't receive a complete transmission, it will not set its interrupt flag ( $RI$ ). If  $RI$  is never set, then the program will loop forever on line 17 waiting for the interrupt flag to be set. Therefore, no more transmissions are initiated and the display updates cease.

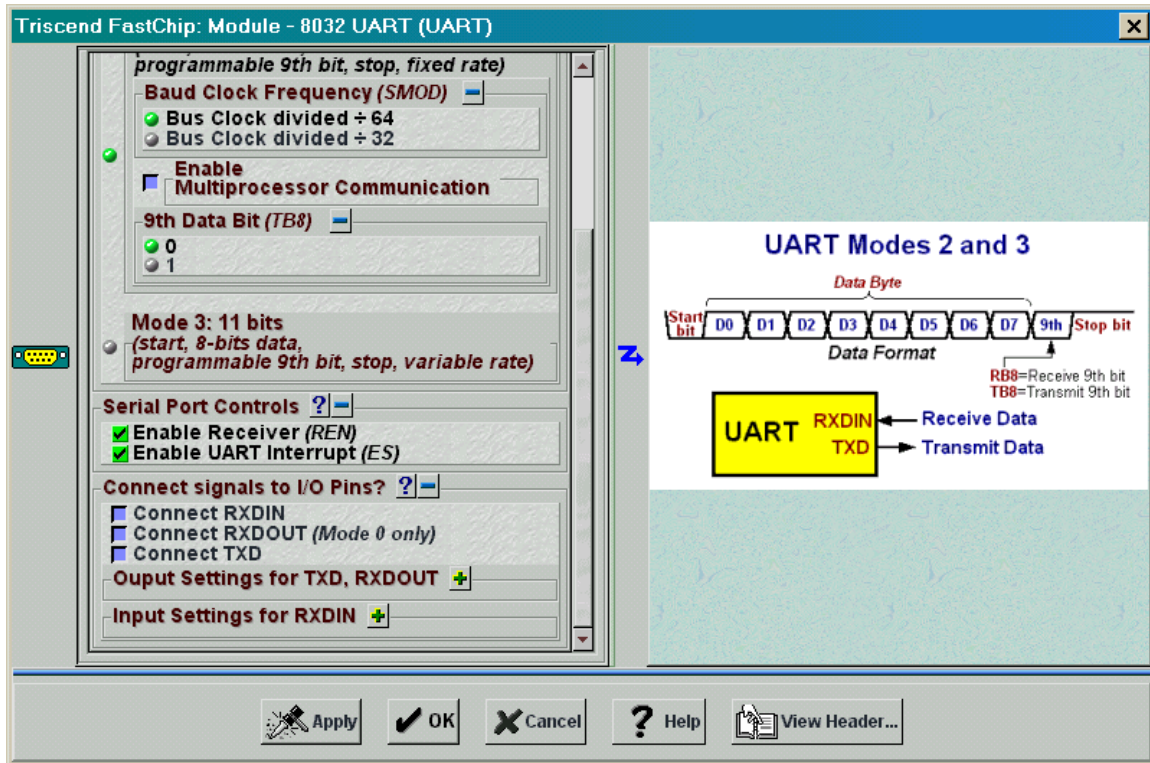
### **Design 2.3 - UART Loopback with Interrupts**

The previous design has the drawback that the program gets stuck in a polling loop once the loopback path is interrupted. The example design in this section fixes that problem by using interrupts from the UART.

Start this design by opening the **Chap22** project with FastChip and then save it as **Chap23**. You don't have to change any of the hardware interconnections, but you do have to enable the 8032 MCU interrupts and the UART interrupts. Click on the Interrupts button in the Dedicated Resources area of the FastChip project window. In the **Interrupt Controller** window that appears, click on the Enable all interrupts checkbox. Then click on OK to finalize this change.



To enable the specific interrupt from the UART, click on the UART button in the Dedicated Resources area and then click on the Enable UART Interrupt checkbox in the **8032 UART** window. In addition to this, click on the Enable Receiver checkbox. This will move the enabling of the UART to the initialization subroutine so you don't have to do it explicitly at the beginning of the `main()` routine (see line 11 of Listing 3). Then click on OK to close the window.



Once you have made these changes to the hardware, you can generate the header file that links the CSoc hardware and software.

Start the coding for the application software as in the previous designs examples by creating a keil folder in the Chap23 FastChip project folder. Then start the Keil IDE and type the code from Listing 4 into a new window. The `main()` routine on lines 9–13 initializes the MCU peripherals and then enters an infinite loop (line 12).

All the real work is performed by the interrupt handler on lines 15–29. This handler is activated when interrupt 4 occurs. What is interrupt 4? Table 8 lists the various interrupt identifiers associated with the various sources of interrupts in the 8032 MCU. Interrupt 4 is associated with the UART. So the `UART_ISR()` routine is called whenever the UART transmitter or receiver generates an interrupt.

Lines 17–21 handle interrupts from the UART receiver whenever it receives a byte of data (as indicated by the `RI` flag being set). The value in the receiver buffer is stored in the `rcvCnt` variable on line 19. Line 20 clears the receiver interrupt flag.

Lines 22–28 take care of UART transmitter interrupts that occur when the transmitter has finished sending a byte of data (as indicated by the `TI` flag being set). On lines 24–25, the last value received by the UART is compared to the last value transmitted. The LED digit displays `O` if the values match (line 24). If the values don't match, the LED digit displays `E` and the error counter is incremented (line 25). Then the transmitter value is incremented and loaded into the transmitter buffer on line 26. The transmitter interrupt flag is cleared on line 27.

Why does the interrupt handler compare the transmitted and received values when a transmitter interrupt occurs and not a receiver interrupt? If the loopback path is broken, no more receiver interrupts occur so any comparisons in the receiver portion of the interrupt handler would never be executed. But the transmitter keeps pumping out data even when the loopback path is broken so the transmitted and received values are always compared and the results are recorded and displayed. Therefore, the status display never stalls.

Once you have the code typed in, save it in the keil folder as `chap23.c`. Then create a new Keil project called **chap23** and add the `chap23.c` file to it. Set the compiler and linker options as you did in the previous designs and then build the `chap23.hex` file. Return to the FastChip project window and do a binding operation on the **Chap23** project. Then download the hardware configuration and the `chap23.hex` file to the CSoc Board.

#### **Listing 4: C application code for the Interrupt-driven UART loopback design.**

```

1  #include "..\Chap23.h"
2
3  #define LETTERE 0x79
4  #define LETTERO 0x3F
5
6  unsigned char txCnt=0, rcvCnt=0;
7  unsigned int  errCnt = 0;
8
9  main()
10 {
11     Chap23_INIT(); // initialize on-chip resources
12     while(1) ;    // wait while interrupt services run
13 }
14
15 static void UART_ISR(void) interrupt 4 using 0
16 {
17     if(RI)
18     {
19         rcvCnt = SBUF;
20         RI = 0; // clear the receiver interrupt
21     }
22     if(TI)
23     {
24         if(rcvCnt==txCnt) ledPort=LETTERO;

```

```

25     else {ledPort=LETTERE; errCnt++;}
26     SBUF = ++txCnt;
27     TI = 0;    // clear the transmitter interrupt
28     }
29 }

```


---

**Table 8: Keil interrupt identifiers for the 8032 MCU in the Triscend CSoC.**

---

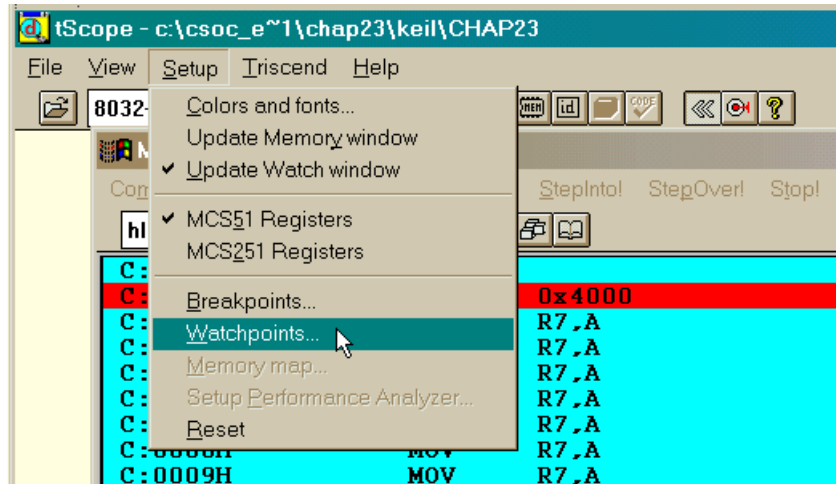
Interrupt source	Interrupt Id
External Interrupt 0	0
Timer 0 Overflow	1
External Interrupt 1	2
Timer 1 Overflow	3
Serial Port	4
Timer 2 Interrupt	5
High-Priority Interrupt	6
DMA	7
Hardware Breakpoint	8
JTAG	9
Software Breakpoint	10
Watchdog Timer	12

Now that your CSoC Board is loaded with the loopback hardware and application code, click on the Run⇒dScope Debugger... menu item in the Keil IDE to start the debugger. Select the 8032-te5.dll entry in the drop-down list near the top of the **dScope** window and establish the debugging link with the CSoC Board by clicking on OK in the **TCP/IP Configuration** window. Next click on the Reset button in the **Toolbox** window. Then load the debugging information from the chap23 OMF file into dScope with the File⇒Load object file... menu item.

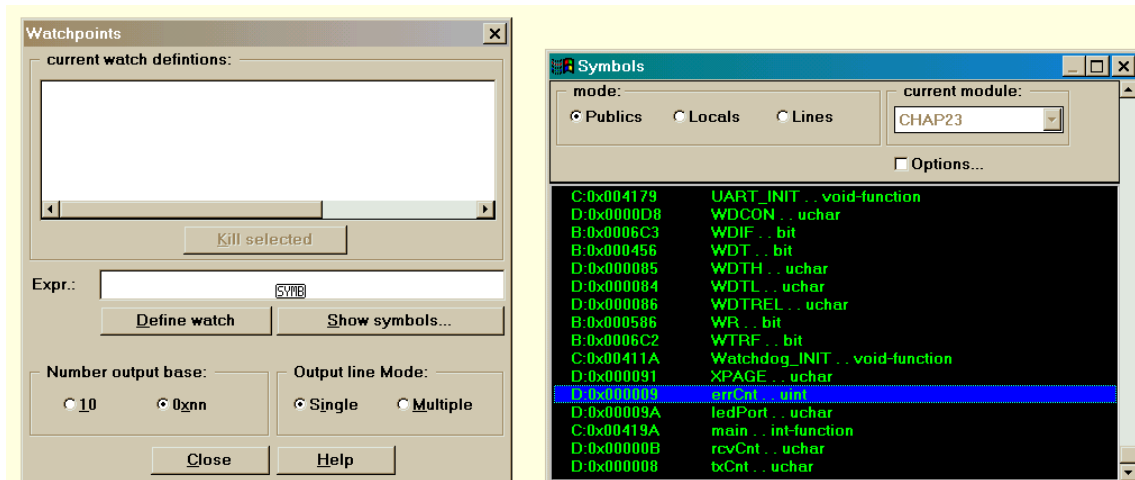
Before starting the interrupt-driven UART loopback program, lower DIP switch #1 into its OFF position to establish the loopback connection. Then click on Go! or the  button in the **Module** window and the program starts executing. You should see O on the LED digit, indicating that the receiver is getting the bits sent by the transmitter. Then flip DIP switch #1 to its ON position and you should see an E displayed on the LED digit as the

loopback path is broken. Finally, lower DIP switch #1 to re-establish the loopback connection and the O will return to the LED digit. So your program now indicates the current status of the loopback path and doesn't get stuck waiting to receive data which never arrives.

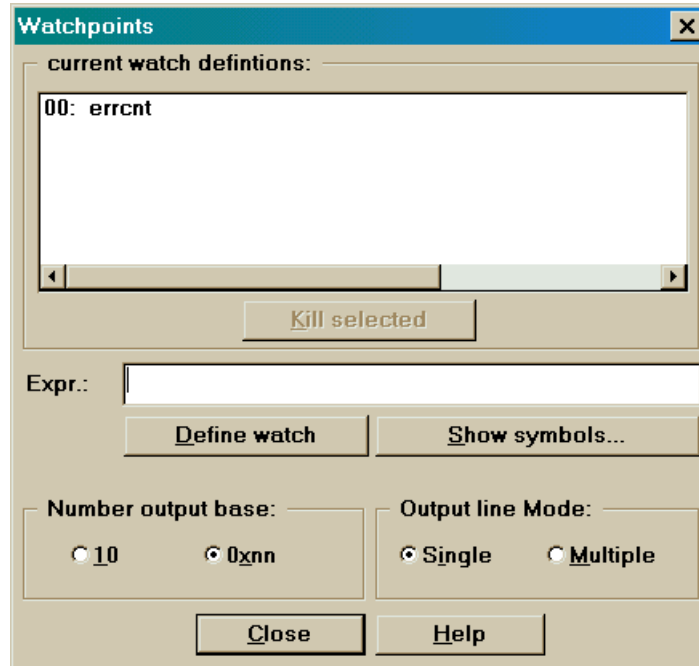
How can you check the number of transmission errors recorded in the `errCnt` variable in your program? The easiest way is to get dScope to query the value of `errCnt` through the JTAG debugging interface. Begin by clicking on Setup⇒Watchpoints... in the **dScope** window. This will cause the **Watchpoints** window to appear.



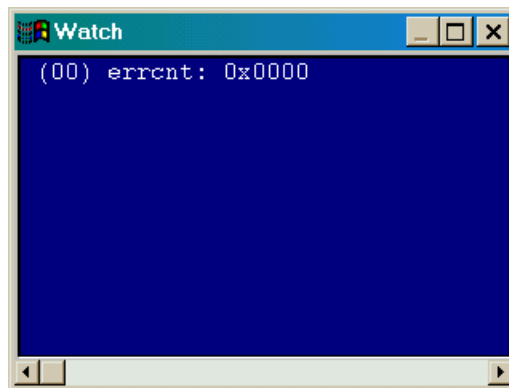
Click on the Symbols button in the **Watchpoints** window to bring up the window that shows all the variable names in your program. Click on the Publics radio button in the **Symbols** window and then scroll down in the list until you find the `errCnt` variable. Click and drag the `errCnt` variable over to the Expr: box in the **Watchpoints** window.



Next, click on the Define watch button and the `errCnt` variable will be added to the list of current items to watch. Click on Close to remove the **Watchpoints** window.

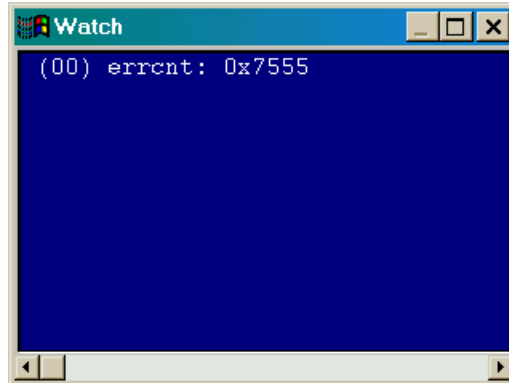


Now the debugger is set-up to watch the value of the `errCnt` variable. `dScope` will query the value through the JTAG interface whenever you halt the program. Click on the Stop! button and then press the Reset button. Make sure DIP switch #1 is in the OFF position. Then click on Go! to start your loopback program from the beginning. The LED should display 0 indicating that the loopback path is in place. Now if you click on Stop! to halt the program, `dScope` will fetch the `errCnt` value from the 8032 MCU and display it in the **Watch** window as shown below. (If the **Watch** window is not visible, click on View⇒Watch window in the **dScope** window.) The **Watch** window reports the `errCnt` variable is set to zero. This makes sense since no transmission errors should have occurred yet.



Now click on Go! to resume execution of the loopback program. Raise DIP switch #1 and then lower it after approximately one second. Then click on Stop! and the **Watch**

window will show the number of transmission errors that occurred during the one-second break in the loopback path.



Does this number make sense? The 8032 MCU is running at 25 MHz and the UART is in mode 2 so it transmits a single bit every 64 clock cycles = 2.56  $\mu$ s. Each byte transmission consists of eleven bits (a start bit, eight data bits, one programmable data bit, and a stop bit) over an interval of 28.2  $\mu$ s. So a one-second interruption in the loopback path will cause  $1 \text{ s} / 28.2 \mu\text{s} = 35,500$  errors. The **Watch** window reports `errCnt = 0x7555 = 30,037` which is pretty close given that you have to manually open and close the DIP switch.

## Design 2.4 - 8032 MCU Memory Tester

Your last MCU-based design in this chapter will test the external SRAM chip on the CSoC Board. It will use the same hardware configuration as the first design (Figure 11). The MCU will determine the segment of SRAM to test by reading the DIP switch settings through port **P0**. Then the MCU will write a pseudo-random series of bytes to the SRAM segment and read it back to see if it was stored correctly. Finally, the result of the test is displayed on the LED digit.

### ***Address Translation in the Triscend CSoC***

In order to understand how the SRAM testing program works, you need to know how the CSoC assists the 8032 MCU with memory accesses. The CSoC has a 32-bit CSI address bus that is used to selectively access various components of the internal circuitry (registers in the CSL, DMA controllers, etc.) or external devices (e.g., memory devices connected to the MIU). But the 8032 MCU uses 16-bit addresses. In order for the MCU to be able to access all the circuitry inside and outside the CSoC, address mappers are provided that translate the 16-bit *logical addresses* into full 32-bit *physical addresses*.

An address mapper uses three parameters to control the translation process: